

Final Report: VizDoom AI via Deep Reinforcement Learning

CARDIFF
UNIVERSITY

PRIFYSGOL
CAERDYDD

CM3203, One Semester Individual Project

40 Credits

School of Computer Science and Informatics

Author: Aaron James
Supervisor: Frank C Langbein
Moderator: Steven Stockaert

Abstract

In this paper, I will adapt an existing algorithm to the ViZDoom visual deep reinforcement learning library and try to find optimal algorithmic settings for some default scenarios provided by the library under the PPO and A2C algorithms. I will attempt to further this into playing The Ultimate DOOM once these settings are found. I will follow existing novel methodologies such as transfer learning, curriculum learning, and reward shaping to allow the machine learning algorithm to develop an understanding of its environment more quickly and to enter complicated scenarios with existing knowledge from previous, more basic scenarios.

Acknowledgements

I would like to thank my friends and family who supported me throughout this project and encouraging me.

I want to acknowledge the help my friends offered me too, such as Chris and Courtney sharing resources on neural networks and reinforcement learning, Kacper and Tom for assisting me in proof reading, Sam and Hannah for being rubber ducks.

Finally, I would like to express my gratitude towards my supervisor Frank who guided me on this introduction to reinforcement learning journey.

Table of contents

Introduction	2
Background	3
Approach	6
Scenarios	11
Justifying Algorithmic Settings	12
Implementation	13
Results and Evaluation	14
Preliminary Testing	14
Basic.py Testing	16
Learning Rate Tests	16
Learning Rate Tests	16
Gamma Tests	19
Gae Lambda Tests	22
Deadly Corridor.py Testing	25
Difficulty Tests	25
Learning Rate Tests	27
Future Work	30
Conclusion	31
Self-Reflection.	31
Conclusion	32

Introduction

This project will focus around optimising a pre-existing deep reinforcement learning algorithm to play a variant of the game The Ultimate DOOM [1] called ViZDOOM. I will do this by using a system that randomly inputs commands to probe the environment and understand what possible and what functions are available to me.

Once the random system is created, I will use it as a basis to expand on implementing a deep reinforcement learning algorithm with emphasis on PPO but some A2C trails are included. These algorithms will enable the system to learn the environment it's in via a Q-learning and actor-critic style of learning. Contrasting the two algorithms will allow me to understand which one is optimal for the situation.

These algorithms will run on the standard scenarios that come with ViZDoom initially to establish good settings. I will investigate the settings of these algorithms to test them. Taking the best outcome at each test will optimise the algorithm towards improving speed of task completion, increase the total reward output, and improve the rate of learning dramatically.

After doing these tests I will use novel techniques such as curriculum learning and reward shaping to develop this solution further into more complicated scenarios such as that of the Deadly_Corridor.py and The Ultimate Doom's levels. Additionally, I will use the scenarios to attempt to pre-train a model to understand some core mechanics of Doom such as health pickups, armour pickups, killing enemies, not missing shots and more.

In this report, the results and evaluation section will be unconventional. I will perform a series of tests, analyse them, do another series of tests, and do an overall conclusion for the set before moving on to a new series. I will go on a scenario-by-scenario basis, then algorithm by algorithm. The series will consist of minor adjustments to a single parameter to trial towards a more optimal parameter set overall.

Background:

First, I searched for a good environment to develop my solution and I found ViZDOOM: a visual reinforcement learning environment for development of machine learning algorithms that was built upon ZDOOM (see reference [2] for ViZDOOM and [3] for ZDOOM). ViZDOOM comes with stock scenarios to train agents with. It also contains tutorials and guides on how to construct an agent class as well as how to use machine learning algorithms.

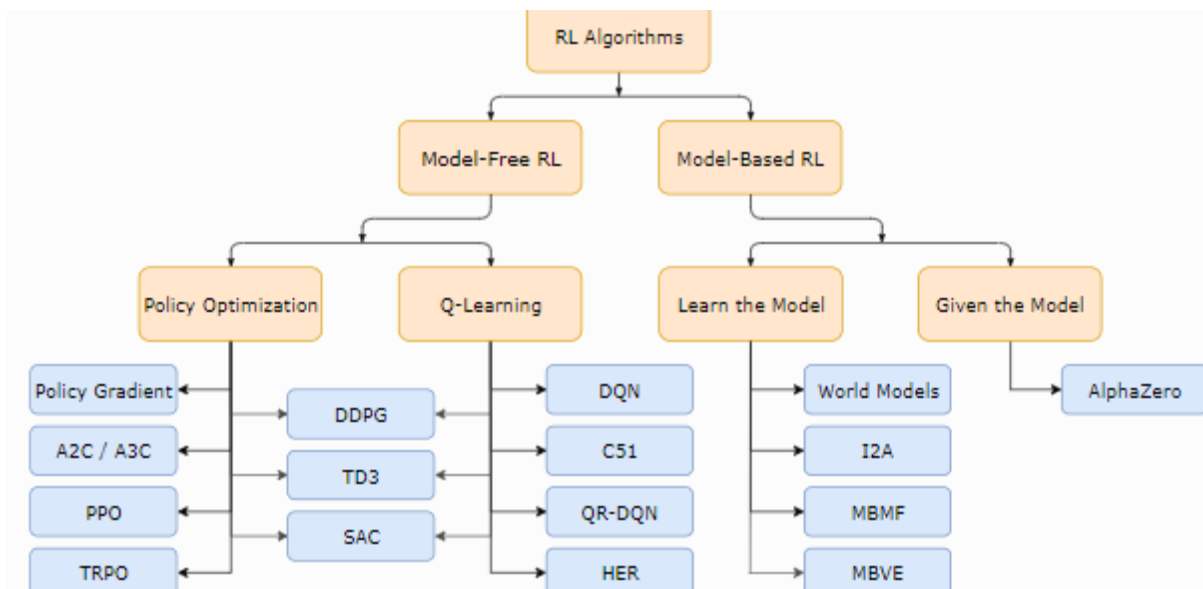
Once I had the environment set up, I did some preliminary testing by trying to produce an agent that could randomly push buttons without any machine learning. I did this to see how the environment would work on my machine. I followed a guide by Michael Kempa (see reference [4]) to study what functions are available in the module.

The preliminary testing looked promising, and the environment began working after dealing with some installation issues.

Now I needed a source of algorithms I could trial. I started to investigate the terms surrounding these algorithms, like Q-learning, actor-critic, the acronyms of the algorithm names, and what makes them unique.

I found a webpage from OpenAI (see reference [5]) that details what different kinds of reinforcement learning algorithms there are. The webpage gave me plenty of other points to

research as stated above. They used a helpful diagram to establish how the algorithms operate in a family tree style diagram, seen below.



The first set of branches covers model-free vs model-based approaches. I had a misunderstanding when looking at these terms before where I thought a model was an environment, but they are different. An environment is the space the agent is present in and can interact with, but the model is the mental map the agent builds up of how this environment works. Not all agents have a model. Model-based agents are more complicated than model-free agents because building this model is difficult and can allow the agent to plan, which is another complicated aspect that can be handled in many ways. Model-based methods do tend to perform better, however. I would use a model-based application so that I could ensure my agent can more consistently beat the levels I choose to give it.

Model based learning algorithms seem much more complicated to implement and rely on having pure planning approaches, or in the case of expert iteration, it follows on from pure planning by learning an explicit representation of the policy function. The agent uses a planning algorithm in this model to generate actions for the plan by sampling from the current policy. The planning algorithm produces better outputs than the policy function, which is then updated following the planning algorithm.

An implementation of model-based algorithms uses a master agent with many sub-agents. The sub-agents learn the environment through experience, and since there are so many sub-agents, the combined experience can be implemented in the master agent to essentially learn from the different sub-agents. I would like to use this iteration if not for the difficulty in implementation of a model-based solution.

I think that I should follow what OpenAI suggests given they made an agent that beat DOTA2 professionals. I will stick to a model-free approach for simplicity.

Down the next set of branches are the umbrella terms for the approaches used when designing these algorithms.

“Policy Optimisation” Methods involve learning a policy directly from experience and forgoing a value function. Policy optimisation substitutes a value function for an approximator. The definitions of a policy and value function will be covered later. The policy learned is a policy that maps the state of the environment without a model to an action. Policy optimisation can be done in various ways.

One such way is called “Gradient ascent”. Gradient ascent takes a function and optimises towards a local maximum but won’t necessarily reach a global optimum. This can cause iterations of the same agent to produce different results. Policy optimisation is performed on policy, meaning each update only uses data collected while acting according to the most recent policy.

Policy optimisation is useful because the algorithm directly optimises for the performance tracking detail. The optimised datapoint can be reward or time left alive. Policy optimisation can be superior to Q learning due to being able to directly optimise towards something. Policy optimisation is generally a more stable method of reinforcement learning.

A simple example of a policy optimisation algorithm can be found at reference [6].

“Q learning” approaches involve learning around a value function and focuses on satisfying a bellman equation, making it data efficient. It is an off-policy strategy, so it learns with all information available. Not just the previous epoch with the latest policy. Q learning focuses on the next step only and can sometimes make decisions that go against its policy, which is why it is an off-policy strategy.

Q learning (see reference [7]) is an unstable practice comparatively to a policy optimisation approach because small updates to the Q function can significantly change the policy and therefore for the future steps the agent can take. Q learning algorithms don’t directly optimise for a data point, and there are many failure points. They can beat out policy optimisation via computation speed because they are simpler to operate since the policy table is much smaller and less relevant. Sometimes Q learning and policy optimisation are equivalent in output. The outputs of both algorithms are dependent on the simplicity of the environments they operate on.

The above link also contains some algorithms that I considered using in my final solution. Moni provided search terms for me to expand my knowledge of the various algorithms that exist like PPO (Proximal Policy Optimisation), A2C/A3C (Asynchronous Advantage Actor Critic) and DQN (Deep Q Neural Network).

The papers involving some of the algorithms above can be found at references [7] and [8].

“Learning the model” approaches learn a model of the environment through play and experience so that the agent can develop their own forward planning methods under the Monte Carlo Tree Search

“Given the model” approaches involve designing a model of the environment myself and handing that model to the algorithm to use. AlphaZero (see reference [10]) for instance uses the model to plan by using the Monte Carlo Tree Search to find promising actions in the future.

I won’t be using the model-based methods because they will be too complicated for my first-time using machine learning, and since I will be mostly using my home desktop computer, I don’t have the processing power nor memory needed for more complicated algorithms to operate. That leaves me with the model-free approaches of Q learning and policy optimisation.

A value function is a method of estimating how good a current or future circumstance is. The value, or advantage of a current state can be discovered given trial and error and by considering the actions possible from the current state or in a future state.

A policy is a table that tracks these state action pairs and stores their advantage so that an agent can refer to this policy when seeking states with the highest advantage.

Both details can be found in reference [11].

A deterministic policy maps state to action without uncertainty which can happen on for instance a chess table. A stochastic policy outputs a probability distribution over actions in each state. This process is called partially observable Markov decision process. In this report, I will not discuss the Markov decision process in detail, but it is a method that is used by many algorithms to decide what action to take next given the advantage of the current and given future states. It is critical for making many algorithms, but in my code, I will not be interfacing much with it as it will be handled for me by a module supplying the algorithms. For more information about the Markov decision process or the Monte Carlo Tree Search, Chapter 3 and 5 of Sutton and Barto's reinforcement learning book (see reference [12]) is great. It taught me a lot of the core aspects to reinforcement learning and gave me search terms.

To help develop this plan into something more, I searched for guides and tutorials on how to implement these algorithms to get an idea of what it would look like on the coding side. I found a great video (see reference [13]). In this video, Renotte goes through the process of setting up the ViZDOOM environment up to reward shaping and additional post-test optimisation regarding parameters, reward shaping, and curriculum learning. All these steps would be useful to me. Renotte also goes through steps I had not considered needing to do, like wrapping the ViZDOOM environment in the OpenGym environment.

I began to search for other modules in Python and found a webpage discussing the top 20 reinforcement learning libraries. I searched the names and found some reputable libraries such as OpenAI Baselines, Stable Baselines, and I investigated TensorForce too since it has support for ViZDOOM. I decided to go with stable baselines because it has TensorBoard support and plenty of algorithms to choose from. Renotte also uses Stable Baselines which could be useful for having a basis on knowing when PPO should operate properly, and I can extrapolate that to other algorithms like A2C and DQN.

For the webpage discussing reinforcement learning libraries, see below reference [14].

Approach

Following my background research, some changes need to be made. Nicholas Renotte has done a brilliant job teaching me via a YouTube Video how to produce a piece of code that does almost what I envisioned of this project in its entirety. Using this as a starting point, I will expand the scope of the project from simply using a reinforcement learning algorithm to produce something that can play Doom and move it on to being competent at doing so and on more scenarios such as those listed in the ViZDoom library provided by Mywldmuch. Given enough time, it could be used on the base game, Ultimate Doom.

Initially I will start by getting a single algorithm working and trying to learn more about how it operates. Then I will expand this library of algorithms to include as many as I can find with modules available to me. After a testing process where I run a basic unoptimized version of the algorithms under the ViZDoom basic.py environment, choose one or two of the best algorithms, then optimise them. The basic.py environment consists of a spawn point in the centre of a room on one side for the player, and a randomly placed spawn point on the opposite side of the room for an enemy. The goal is to move left and right, then shoot the enemy once. After the algorithms have completed training on basic.py, I will trail other environment models provided, trying out reward shaping and curriculum learning where the algorithms struggle. Then I will try to use transfer learning to gain skills for the main doom game, which is where the project will be completed.

I will also adjust the modules I am using for this project. I was going to use matplotlib to produce some graphs to analyse the performance of the algorithms to compare and then finally to show how a competent AI learned to play. Instead, this functionality comes mostly inbuilt with TensorBoard - besides a few potential graphs for tracking data used to indicate how well the algorithms do compared to one another. I will still use matplotlib to produce a graph potentially showing number of enemies killed, total successes, speed of successes and the like, but that will be after getting a functional algorithm together.

I will use pytorch since there is a great module built on top of it called Stable Baselines 3 which uses some prebuilt algorithms such as PPO, A2C, and DQN among others. Using these, I will be able to easily swap them in and out of my program or run them in parallel to see what the differences are. Running them sequentially or in parallel easily is a must for me since I am inexperienced in the AI field and will allow me to compare more easily and study what analytics matter most. Additionally, since they mostly use the same hyperparameters, getting unique settings together for each is simple.

The only other notable module that I will be using is OpenAI's gym which will be used as a wrapper for the ViZDoom environment to use some inbuilt functionality to gym's environment. Gym's core functions help the program run more smoothly with managing rendering, on-input calls, and resetting the game state to the start for a new episode.

With respect to finding which algorithm is initially optimal, I must find which algorithms have a Discrete action space, provided by stable baselines 3. This action space is in relation to binary inputs. For instance, left is either pushed or not pushed. Some algorithms require a continuous action space, such as that of an analog stick. I will not be able to use these following Nicholas Renotte's solution. I may come back to implementing them. This narrows my search down to DQN, PPO, and A2C.

With the algorithms implemented I will be able to run the written code and establish details including which algorithm runs faster, which algorithm learns the fastest, which algorithm converges the soonest, and finally, which algorithm uses the least memory. The data will be selected from the TensorBoard log output of the program running alongside the learning loop. With this information I can decide on what algorithm would be the best use of my time to move forward with. I could also rank them so should something go wrong; I can return to this ranking and select a second-place algorithm to trial where the first may fail.

I will continue to trial this algorithm on various scenarios to see how it holds up against different environments. Initially, I will introduce the algorithm to each new environment with no prior memory. If it struggles, I will lower the difficulty of the level and improve how the reward is distributed to the AI to encourage it to take steps closer to a goal, shoot at enemies more frequently, stand still less, and speed up completing the goal.

I will be adjusting several hyperparameters to encourage the machine learning to explore more. I had a discussion with my supervisor Frank Langbein about the hyperparameters offered by the PPO and A2C algorithm where we discussed at length the explanations offered by the stable baselines 3 documentation and what they mean as they aren't self-identifying variables without prior knowledge. I have chosen the hyperparameters below since I understand vaguely understand them more following our discussion, and they are easy to systematically adjust and log improvements. When I get the data to converge initially during preliminary tests, I will begin to adjust individual hyperparameters in arbitrary but similar increments according to their default values. Some data,

such as learning rate, begins very low. I will increase this by at most 0.0002 between tests. Other parameters begin at 0.5, which will change by around 0.1 per test.

For documentation for stable baselines 3 [extra], see reference [15].

The details of the discussion are as follows:

Learning rate

Learning rate a value that controls how well new information is learned vs the old information being kept. A high learning rate causes new information to have much more importance than old information. It is useful to keep learning rate low and increase iterations because it allows agents to explore the environment aptly while still learning information that can improve the agent's reward. I chose to use the values 0.0001, 0.0003 and 0.0005. I chose these values because tiny increases in learning rate can cause substantial changes in output as seen in the graphs above. This increase is linear, easily comparable, and will give me an indication of how much of an impact learning rate has over the learning operation.

Gamma

Gamma is a discount factor between the range 0 and 1 relating to how relevant the past is to the model. Over time, the model begins to 'forget' or considers information far in the past as less important. As standard, this is around 0.8 or 0.9. For PPO and A2C, this is 0.99 by default. Having a high Gamma is useful in circumstances where greedy approaches are beneficial. A greedy approach is an approach which always takes the best available series of actions to a solution. Gamma can be lowered to cause the model to forget the past more, forcing exploration to refresh this forgotten information. In some cases, the model will take a different action. In others, the model will find the weight attached to the action to be inaccurate. Refreshing this action causes the weight associated to adjust and making it more accurate over time.

Gae_Lambda

Gae_Lambda refers to a multiplier for the generalised advantage estimation (gae) calculation (see reference [16]). Gae aims to measure the advantage given by choosing a particular action in a particular state. Advantage is a measure of how good a certain action or set of actions in the current state is. Gae_lambda acts as a multiplier for the above Gamma in the gae equation, so it serves a similar purpose to Gamma. The values are 1.0 in A2C and 0.95 in PPO by default.

Ent_coef:

The ent_coef or entropy coefficient is a multiplier for the loss function (see reference [17]). The loss function is an equation that maps a real number, a cost, to an action in a state. To achieve an optimal solution, the loss function needs to be minimised. Entropy in this case relates to how difficult extracting information is from data.

By this reasoning, the entropy coefficient should relate to how random the information is deemed to be. The information on screen to make decisions with is not random. For example, if there is an enemy on the left side of the screen, it should remain there until action is taken to shoot the enemy or retreat or something else. Either way, it is a controlled environment, and the data onscreen contains lots of information. Therefore for mostly visual learning I think this variable should remain

very low though I will test how increasing this variable affects learning. In PPO and A2C the value is 0.0.

Vf_coef

The Vf_coef a value function coefficient for the loss function according to stable baselines 3. I am unsure what it represents but this is the larger multiplier in the loss function. Since the documentation is not great for this variable, Langbein and I speculate that it is a variable multiplier for the weight of taking an action. Usually it is around 0.5, in PPO and A2C it is 0.5.

Max_Grad_Norm

The max_grad_norm is the maximum value for gradient clipping. It is 0.5 in PPO and A2C. When optimising, it avoids moving the value of a weight associated with an action too much in one iteration. This act of capping the maximum change avoids the algorithm finding a local maximum reward for an optimal solution and having a hard time getting out of the poor local maximum. Hopefully this being lowered encourages finding a more consistently good local/global maximum reward average. Reward is a numerical value, similar to a score system, that is given to the model when it takes good actions and deducted when it takes bad actions for the circumstances it finds itself in. By lowering this Max_grad_norm, I am hoping that the convergence to an optimal solution will become slower, but it won't overshoot the correct values on the weights so that in a greedy scenario, it will find a true optimal solution.

Use_rms_prop

By default this uses an RMSprop optimiser, and can be toggled to use the Adam optimiser. An analogy given to me by Langbein is as follows:

The RMSprop optimiser can seem like a ball on a slope. When the ball moves downhill, it accelerates until it enters a trough. When any uphill gradient is reached, the ball will stop in place under RMSprop. The Adam optimiser introduces a form of momentum, whereby entering this trough and hitting an upwards gradient does not stop the search of a global optimum. Instead, this momentum can carry the ball over the peak of the next hill, into potentially another, deeper trough, finding an even better minimum. This scenario can be inversed graphically to find maximum reward solutions, which is what my solution will attempt to do.

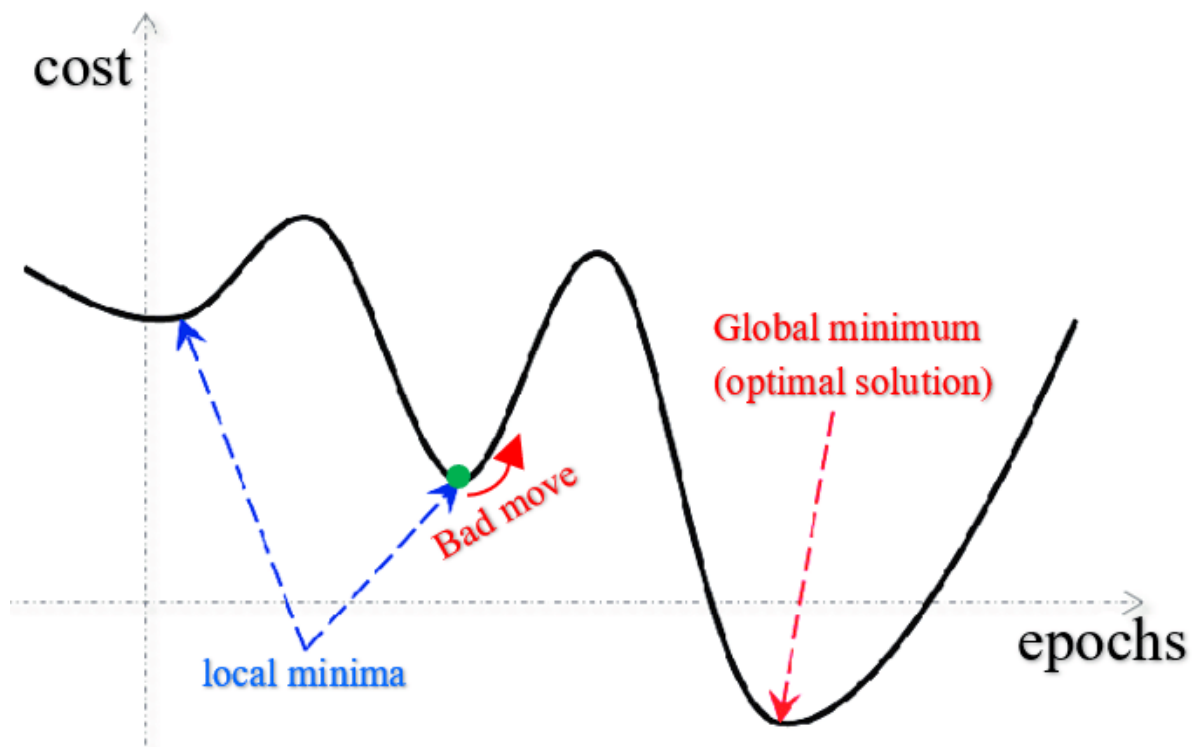


Figure 1: Local minima and global minimum rewards

Figure 1 shows this analogy in more detail (see reference [18]). The “bad move” here will be made by the Adam optimiser. While it is a bad move in the short term, in the long term it may cause the ball to move over the next peak into the global minimum trough. Diagram sourced from:

Rms_prop_eps

Rms_prop_eps is usually 0.00001 in PPO and A2C. This value is supposed to stabilise the square root computation in the denominator of the RMSProp update and is not used when the Adam optimiser is active. Best not to play around with rms/sde values too much as they should remain very small, and I don’t fully understand what the values are for.

Sde_sample_freq

the sde_sample_freq is a value that determines how random exploration is by adjusting the probability over sampling the policy. The policy of a model is like a list of states that are bound to a set of circumstances. Sometimes it is set, so an action has a given weight in a specific scenario making many values discrete, sometimes this is a gradient making all values continuous. Both PPO and A2C are policy gradient algorithms. This value is usually 0.1 in A2C and PPO.

Normalise_advantage

This Boolean value dictates whether to normalise the advantage from the gae. If the output value is small, it can be normalised to be much larger than it initially was. Usually, these algorithms don’t normalise, but since I can’t see this value during runtime, I will need to experiment to see if adjusting this value yields better results.

Once the algorithm learns how to beat these scenarios, the goal will be to translate this newly learned policy to other scenarios. Hopefully, this translation of the same AI to different tasks will culminate in it being able to identify what to do when being dropped into a level. For instance, if the AI beats `DeadlyCorridor.py` (which consists of 6 enemies in a corridor and a piece of armour on the opposite end) the AI should learn that picking up armour will yield a big reward long term and improve its health's effectiveness. Do the same with health pickups, then with shooting enemies, then with navigating to a goal, I could slowly build up a policy that stands a chance against the full game.

Once the AI is competent enough to beat the game, it can train until the project terminates at which the data can be analysed at various points. Looking at around ten thousand iterations, fifty thousand, and then up to its completion should yield an interesting learning dynamic. The TensorBoard log should have a huge, consistent range of information over potential days of runtime if completed quickly enough. The data yielded should be more than enough to conclude the project on a positive note of either success or as a learning experience.

Scenarios:

Basic.py

The `Basic.py` scenario features an enemy spawning randomly on the opposite side of the room to the player. The player can move left, right, and shoot. The player has 50 bullets but can only shoot around 28 in the 300 frames they have to kill the enemy. Once the enemy has been shot a single time, the episode concludes in a success. After 300 frames pass, the episode is considered a failure and concludes.

In `Basic.py` I expect to see that length and reward have an inverse relationship that should resemble each other extremely closely, as when the task is completed quickly, reward should be high. When completed slowly, due to the idle timer and living reward punishment of -1, every tick spent alive will cause reward to decrease. Longer times spent in the scenario will correlate to a lower reward because of this punishment.

Rewards are allocated as follows by default:

Action	Reward
Kill enemy	+100
Miss shot	-5
Spend a frame idle (not killing the enemy)	-1

Deadly Corridor

Deadly corridor consists of a long corridor with 6 enemies flanking the sides of the room. Each set of 2 enemies are stowed away in crevices in the wall like a pocket, that serve to kill agents running directly to the goal: an armour pickup. Picking up the armour will yield a huge reward, but the problem is finding a good path to it, since the enemies will normally kill the agent running directly to it.

In deadly corridor I expect to see length and reward have an inverse relationship since picking up the object quickly will cause the agent to have lost less health or have died less frequently in its series of tests.

Currently, the rewards look like this:

Action	Reward
For moving closer to the armour pickup	+dx
For moving further from the armour pickup	-dx
For picking up armour	+1000

Justify Algorithmic choices:

There are some core limitations to the algorithms I can select from the Stable Baselines 3 [extras] package that I did not catch until I was well into the project. Mainly the input methods. Some algorithms operate on an analog stick's continuous inputs making them unfit for my purposes. There are only 3 algorithms that meet the criteria of taking discrete controller inputs: PPO, A2C, and DQN. I would like to test all of these, but I am limited on time so I will avoid using what I find to be the weakest algorithms.

PPO and A2C were recommended to me by my supervisor who has background in reinforcement learning. They stated that actor-critic approaches yield higher quality results than other approaches. DQN is a stretch algorithm if I have spare time to evaluate more algorithms.

I initially chose to work with PPO as it was recommended to me by my supervisor and Renotte uses it. I could use renotte's video in conjunction with the PPO algorithm to ensure a smooth set up phase for the project, which should give me good grounding to expand upon.

Implementation

I began by following Nicholas Renotte's ViZDoom guide to produce an unoptimized piece of code that can learn a basic scenario. I made a few changes to his code mostly for demonstration purposes. For instance, Renotte implemented a grayscale function using CV2. I too have implemented all the code required just in case the program runs slowly, but I chose to not run it for the time being. Having a grayscale image reduces the memory usage by two-thirds which is great for runtime since there is only 1 colour channel that the AI must study. Additionally, Renotte resized the image to reduce the number of pixels, also reducing computation. I found that when full screening this environment for viewing, the visuals are quite pixelated which I don't like. I have also implemented this but I'm not currently running it for my experimentation. Should computation speed become an issue, I will run these sections.

I did some trials at this point to see how the Stable Baselines 3 PPO algorithm works on basic.py and how long it would take to converge. I found that after around fifty-thousand iterations, the AI would converge to an optimal or near optimal solution under Renotte's parameters. I trialled some other algorithms offered by Stable Baselines 3 such as A2C and DQN without changing the parameters, but I encountered some issues.

A2C converged early but at a low reward and seemed to optimise towards doing nothing under Renotte's parameters, and DQN required 200Gb of RAM which I could not provide for the observation space. I do not know why DQN required this much even though nothing had changed, and I don't know why A2C optimises towards a negative reward (even though that negative reward can go lower than it is). I suspect I can solve the A2C issue by encouraging exploration. The graph always trends downwards from the start and never climbs. There was only 1 occasion out of 17 tests where the data fluctuated frequently but remained between -60 and -180. I began my official tests after this set up phase.

Tensorboard was working well for displaying all the analytics about runtime and had much more data than I was expecting, like clip fraction, policy loss, explained variance and more. Plenty for me to analyse in the results and evaluation, though it might be out of the scope of my understanding. I could use the data in Tensorboard to better tune the parameters to the ViZDOOM environment, but it would take too long to build the experience needed to understand what the values mean well enough to better optimise parameters without trial and error.

Results and Evaluation:

Basic.py tests:

A2C Default Parameters:

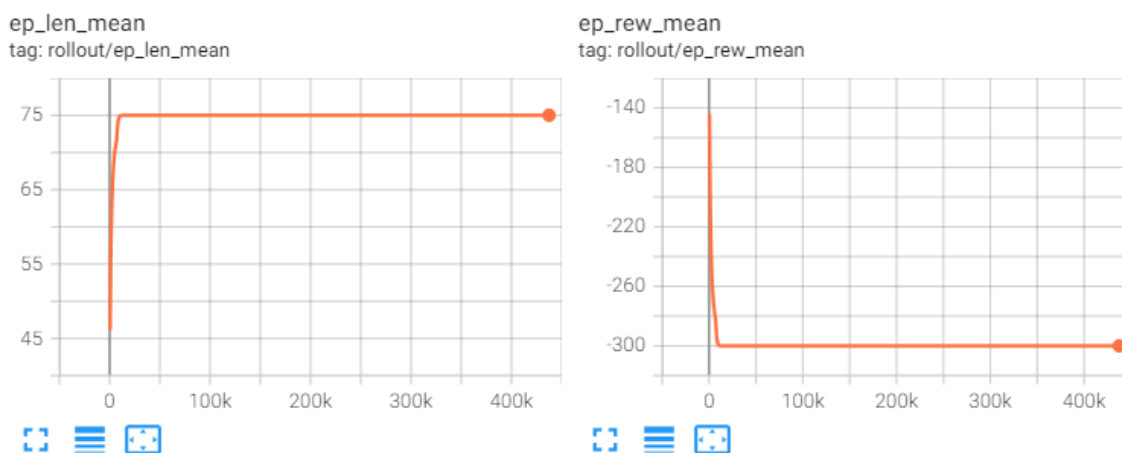


Figure 2: A2C with default settings

The titles for Figure 2 are the `ep_len_mean`, which is the mean length of an episode, and `ep_rew_mean`, the mean reward of that episode. An episode is a single iteration of the environment.

In figure 2, the A2C algorithm with default settings started at a low reward but plummeted to -300 in 4096 timesteps. At this point, the reward plateaued. This was a disappointing result for the first run of the A2C algorithm, but it is an unoptimized solution. Still, I can't understand why this is deemed a maximum for the reward.

The maximum length is achieved by failing to kill the enemy, as explained in the Basic.py scenario above. An optimal reward tends to be around 88+. This is because sometimes the enemy can spawn on the far left or far right of the room. The AI must then move to meet the enemy, then fire. This seems to take on average 12 frames, though I am unsure how many frames are required to touch the left or right walls. Nonetheless, the fact the reward remains at -300 tells me that the AI deems the optimal solution is to sit still and not to fire. Even over a very long iteration period of 400k timesteps, there is no improvement. There is not enough exploration being encouraged.

PPO Default Parameters:

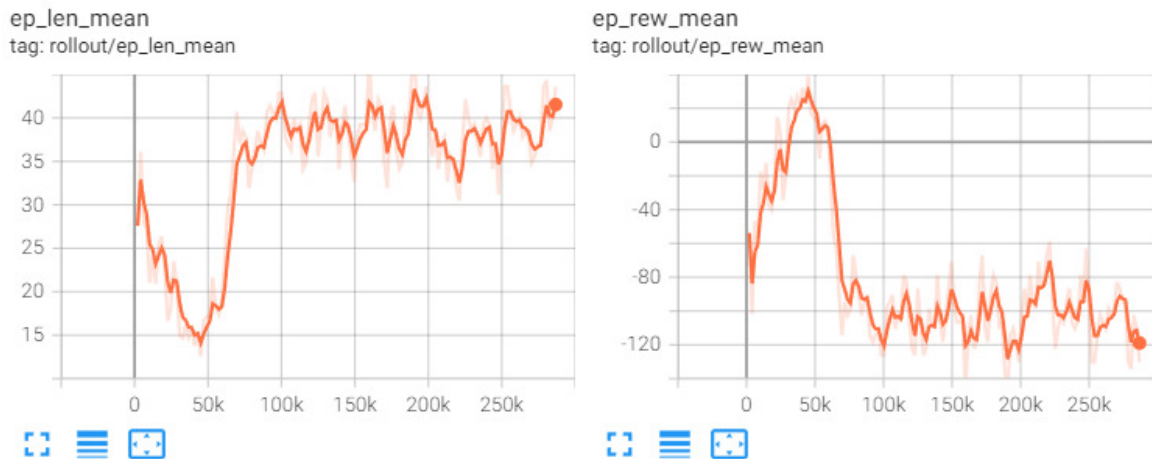


Figure 3: PPO with default parameters

To contrast with the A2C algorithm, PPO in Figure 3 was more volatile and, notably after around 70000 timesteps, the volatility decreases. This means the PPO algorithm is much more successful overall than the A2C algorithm but doesn't converge at all in 4 hours or around 300000 timesteps. PPO shows more promise than A2C in this regard, but more optimisation needs to be done for the graphs to converge.

Figure 3's reward starkly increases and then plummets from a reward of 37 to a reward of -120 at its minimum. The graph does not plateau and still has moderate volatility post plummet. I am running these tests for far too long as they do not yield beneficial results after around 100000 timesteps and I have not encouraged exploration to aid in rebounding reward upwards later on.

PPO Learning rate 0.0001 n_steps 2048:

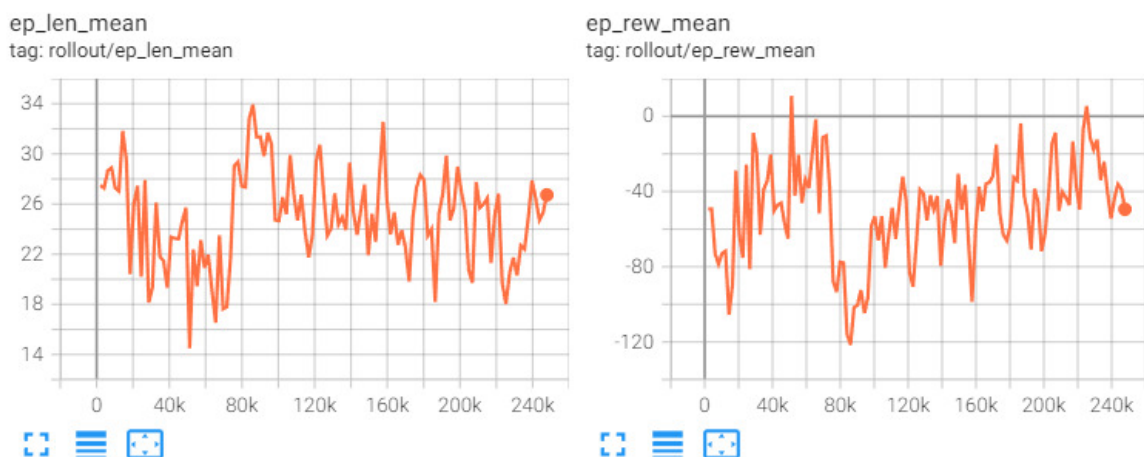


Figure 4: PPO with hyperparameters learning rate 0.0001 and n_steps 2048

Figure 4 shows a hugely volatile PPO algorithm with a maximum. reward of 12 and minimum reward of -120. There is no convergence but there is a general upward trend besides a substantial drop at 68. I think lowering the learning rate was a positive change, but it was not impactful enough.

A2C Learning rate 0.0001 n_steps 5:

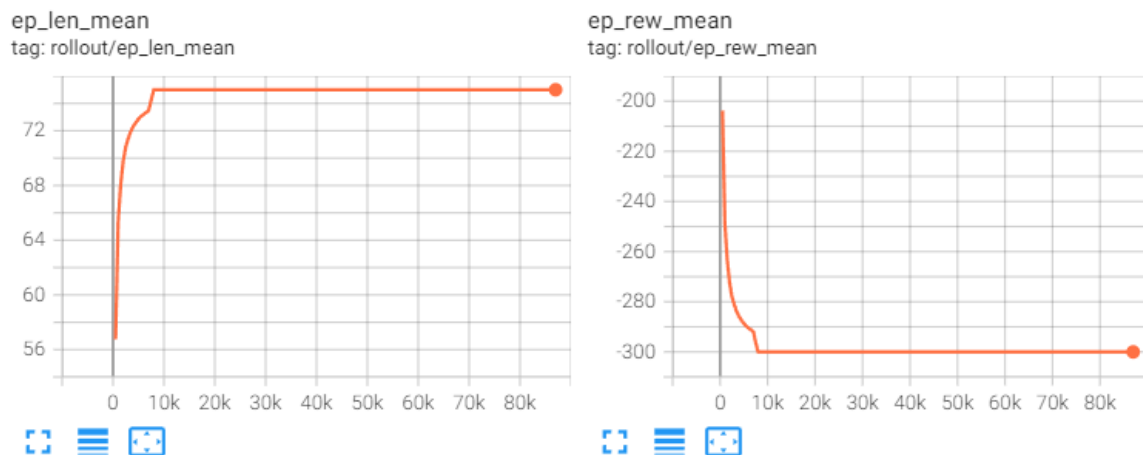


Figure 5: A2C with hyperparameters learning rate 0.0001 and n_steps 5

Figure 5 shows that decreasing the learning rate has staved off the convergence at -300 for around 4096 more timesteps but did not change the results more than that. I concluded this test early due to no changes for a prolonged period.

Reworked learning rate tests:

I conducted a test with the new settings suggested by Langbein. I first adjusted the learning rate to 0.0001 and raised it by 0.0002 increments until 0.0005. The learning rate needs to be kept low to ensure the agent doesn't learn incorrectly too quickly, and to ensure sufficient iterations are done to maintain the information it slowly learns. If the agent learns too quickly, it could associate information incorrectly to reward increases and optimise towards doing something poorly.

I stopped using A2C at this point. Too many issues arose with testing and wouldn't converge no matter what parameters I used. PPO was more stable, easier to work with, and had better documentation so I chose to work with PPO for the remainder of the paper.

Learning rate 0.0001

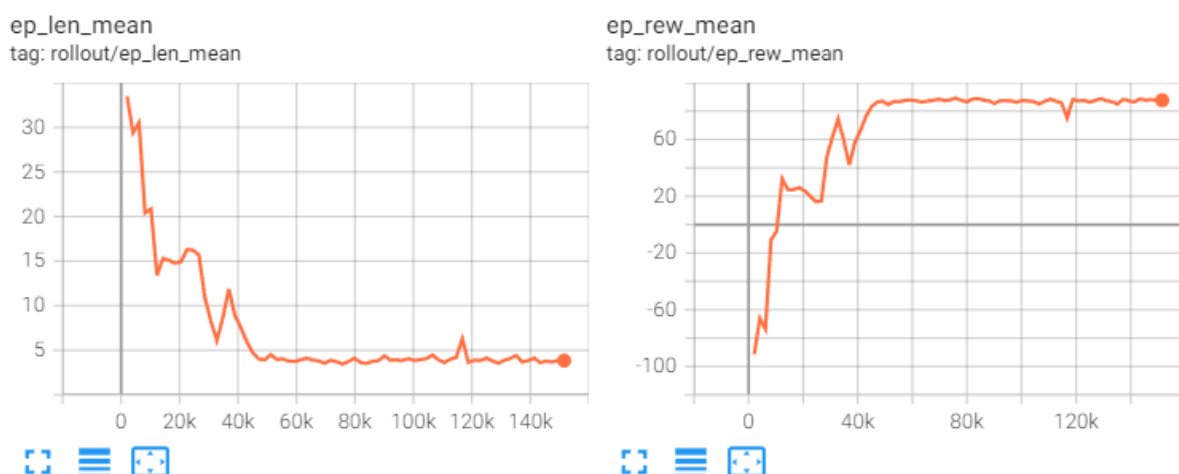


Figure 6: PPO with learning rate 0.0001

At learning rate 0.0001, I saw a regular increase towards a convergence at around 50,000 timesteps. It plateaued at 45,000 timesteps. To plateau means to remain at a relatively constant reward for 20,000 timesteps. For Basic.py, I deemed relatively constant as having a range of 10 for the reward. Timestep 45,000 was the first point where the range thereafter was maintained, where it remained between 83.19 to 87.82. Looking at the gradient at the start of the graph allows me to glean the rate of learning of the model over timesteps. In learning rate 0.0001, this gradient was:

$$(83.19 - -91.31) / 45060 = 0.0038726 \sim 0.00387.$$

The convergence rate (reward gained per timestep) was 0.00387. Using this metric, I will be able to compare meaningful learning speeds with respect to timesteps between algorithms.

Looking into the plateaued range (the range of data after an algorithm is considered “plateaued”), I can see that reward varied from a minimum of 75.19 to a maximum of 89.24. This is a range of 14.05. This range is substantial for a converged solution, but on a macro scale the drop to 75.19 is sudden. This can be seen at timestep 116,700. I attribute this to the exploration and randomness of the scenario. An enemy spawning more on the far left or right of the room will cause short term decreases in reward as is apparent there, but due to it being substantial from 85.86, I think the exploration section of the algorithm attempted something new. This drop in reward is likely a mix of these two factors and will be explored more in future tests on how forgetfulness can be manipulated to encourage exploration.

Learning rate 0.0003

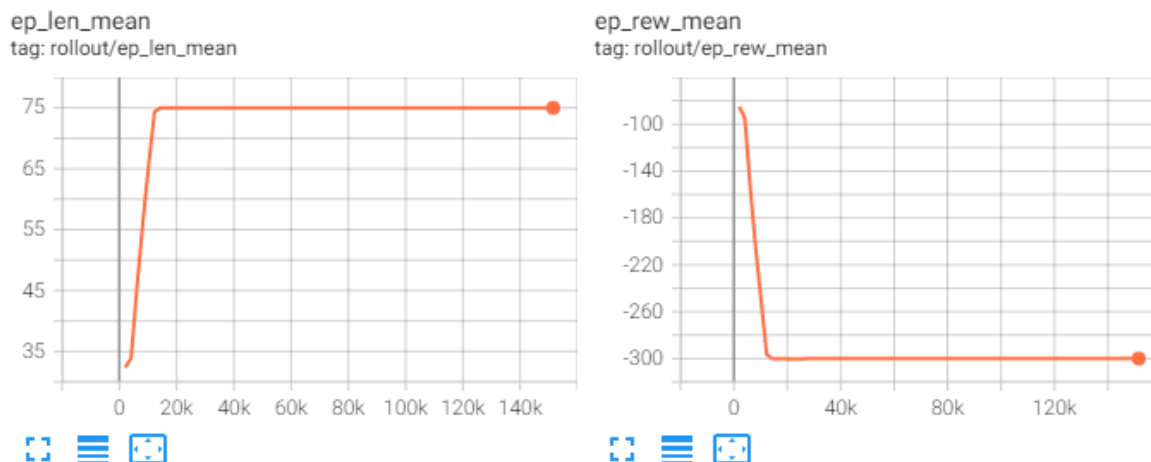


Figure 7: PPO with learning rate 0.0003

For learning rate 0.0003, the result was remarkably different. From the first data point, the reward took a steep dive towards -300. This is an interesting reward for this scenario because it is the minimum earned by doing nothing. This tells me the algorithm established shooting was negative and so could not unlearn that due to exploration not being supported well enough.

The convergence rate of this graph was:

$$(-296.4 - -85.08) / 12,290 = -0.0171944 \sim -0.01719$$

A stark decrease from the previous convergence rate of test learning rate 0.0001.

Learning rate 0.0003 plateaued at timestep 12,290 and had a plateaued range with minimum -300.5 and maximum -296.4. this was a range of 4.1. This smaller range is evident of small change and low entropy. The randomness of the environment and agent, especially when the agent decides doing nothing is optimal, has no effect on the reward so it remains constant.

Learning rate 0.0005

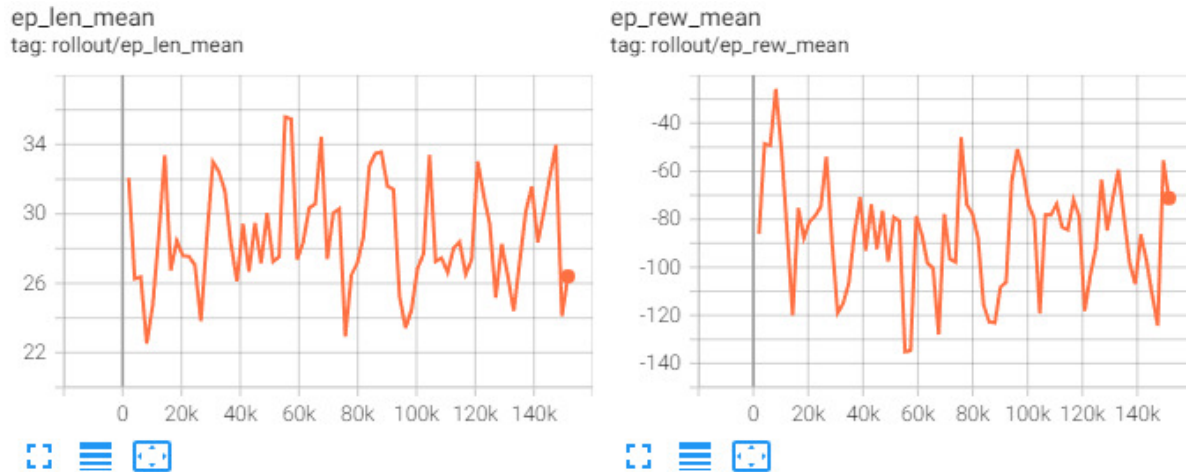


Figure 8: PPO with learning rate 0.0005

For learning rate 0.0005, the graph never converged. It fluctuated wildly over the range of -135.2 to -25.73. This test was quite interesting but difficult to draw metrics from due to the non-convergence and huge variance throughout the graph.

Learning Rate Conclusion:

A lower learning rate yielded much better results. It had a wide plateau range but converged with a mostly positive gradient consistently. It reached a convergence within one-third of the runtime, giving plenty of room for the agent to forget information and relearn its surroundings. The tests have indicated to me that there are several local maxima, and one of which involve doing nothing which is unfortunate. In future, I need to encourage more exploration so that the agent can aptly learn the environment and find a local optimum that isn't -300.

Moving forward, I will use a low learning rate and in more complicated scenarios, reduce it further as additional inputs are required, and the environment becomes more complex. These changes could include pickups, turning/aiming simultaneously with moving or more enemies.

I need to avoid settings higher than a learning rate of 0.0003 as that setting and above will likely cause the agent to struggle to learn properly as seen from the graphs. Additionally, I am unsure if learning rate 0.0003 is strictly inadequate for every setting but it seems like a poor setting for the agent. I think 0.0003 should be explored further given additional time since it may have just been an unfortunate event finding the local maxima at -300. My preliminary testing showed results at this range several times for many learning rates.

Gamma Rate Tests:

Following my learning rate tests, I investigated adjusting Gamma to help encourage exploration. I chose to adjust the data from a suggested maximum upper bound of 0.99 to 0.9 in increments close to 0.05. With a small range of 0.8 to 0.99 to explore, I thought the variables selected were sufficient. For these tests I kept learning rate fixed at 0.0001.

Gamma 0.9

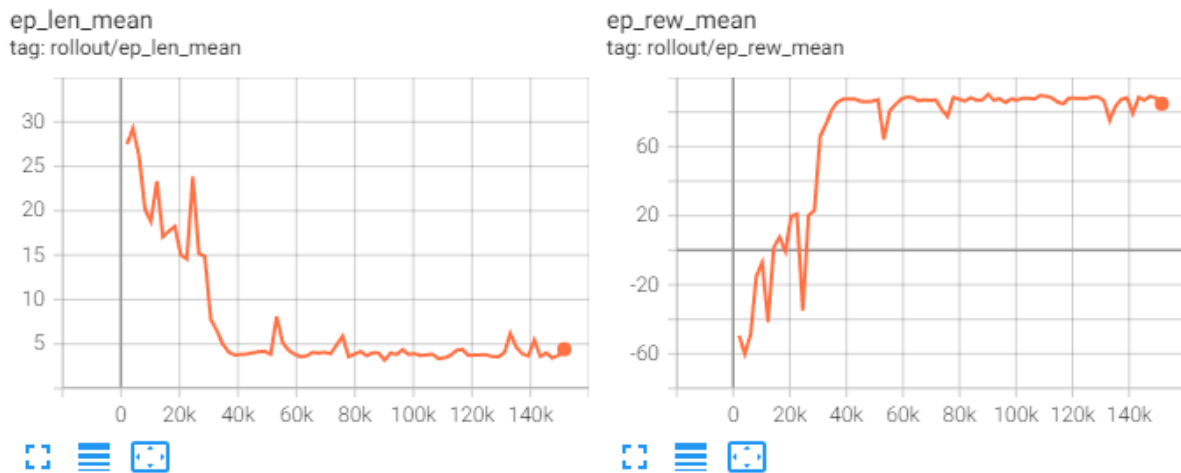


Figure 9: PPO with Gamma 0.9

Decreasing Gamma has the effect of introducing more variances as seen above on the ep_rew_mean graph. Increasing Gamma causes the agent to discount and forget the past more, causing it to explore paths more as seen by the sudden drops in reward. The graph still converges with a learning rate of 0.0001 at around the same time which is good news. This means Gamma is a safe way to introduce or amplify the explorative aspect of the agent.

The Gamma 0.9 agent plateaus at 55,300 timesteps. While this is late for the plateau compared to a normal learning rate, this late plateau was caused by the first sudden drop going down to a reward of 64.18 at timestep 53,250. The maximum reward for the plateau at 55,300 was 88.65 and the minimum reward was 77.39 representing a range of 11.26. A tighter range than learning rate 0.0001 but the variance had already pushed the agent out of plateau range missing a great convergence timing of 34,820 timesteps.

The convergence rate at a plateau of 34,820 timesteps was:

$$(81.14 - -49.32) / 34,820 = 0.00374669 \sim 0.00375$$

This potential plateau convergence rate was lower than the test at learning rate 0.0001 but only marginally, by 0.00008. I believe that the lower gamma would have surpassed the learning rate 0.0001 test if only it started at a lower reward, as that is skewing results to be lower than they potentially should be.

The plateau range for Gamm 0.9 had a maximum of 90.28 and a minimum of 75.18 giving a plateau range of 15.1. This range is wider than the learning rate 0.0001 test which is expected. I decreased Gamma which caused the agent to forget the past more. This should increase variance as important information is lost regarding how to gain higher reward in the hopes that negative actions are forgotten, and positive actions are relearned which was seen in the graph above where the agent quickly reconverged after a substantial drop in reward of around 15. Lowering Gamma had an

overall positive effect, but it skewed results towards seeming more negative. I should not ignore the fact that the agent could forget some critical information but was able to relearn it quickly. That is a desirable and valuable feature for more complicated environments. Final thing of note about this test is it achieved the highest reward seen so far, going above reward 90 reaching reward 90.28 at timestep 90,110.

Gamma 0.95

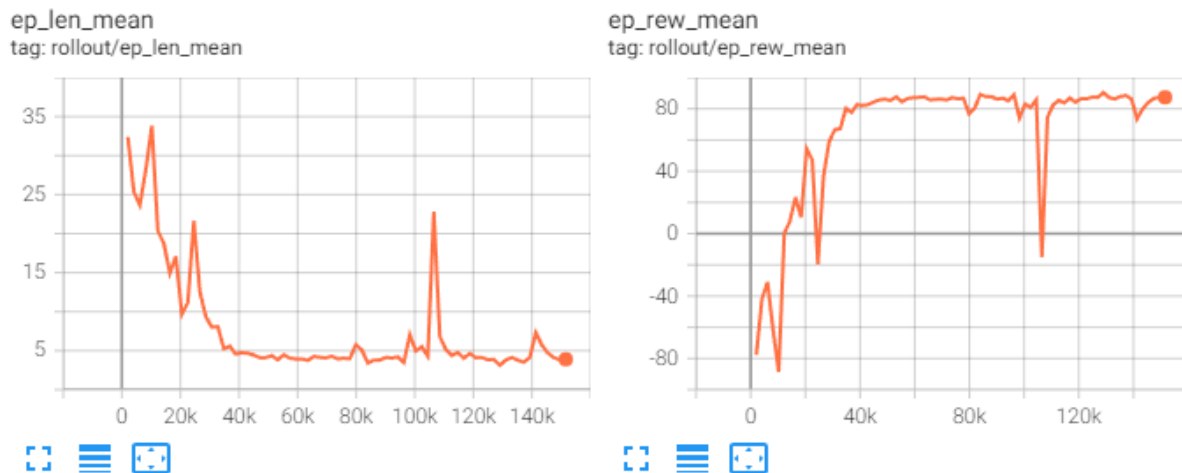


Figure 10: PPO with Gamma 0.95

For the default Gamma, 0.95, there was much more variance compared to the learning rate 0.0001 test. I'm not sure why as this level of Gamma was present in that test too. The gamma 0.95 test plateaued at timestep 34,820 with maximum reward 87.64 and minimum reward 80.31 which is a tight range of 7.33.

The convergence rate of test Gamma 0.95 was:

$$(80.31 - -77.67) / 34,820 = 0.0045370 \sim 0.00454$$

This convergence rate represents the highest convergence rates achieved thus far.

The plateau range for this test had a minimum of -15.02 and a maximum of 90.41, which is a new upper bound. Discounting the extreme minimum, the next minimum is 73.46 leaving a plateau range of 16.95. The new upper bound of 90.41 was achieved at timestep 129,000. Even after such an enormous drop in reward, the agent recovered immediately in the data point after. Quite impressive. Having any sort of Gamma seems quite beneficial to the longevity of the agent and preserving its ability to do well while simultaneously trying to erase negative actions learned. I think future tests on more complicated environments will entail trial and error to find a sweet spot for the environment, but I would guess it will generally lay between Gamma 0.9 and Gamma 0.95.

Gamma 0.99

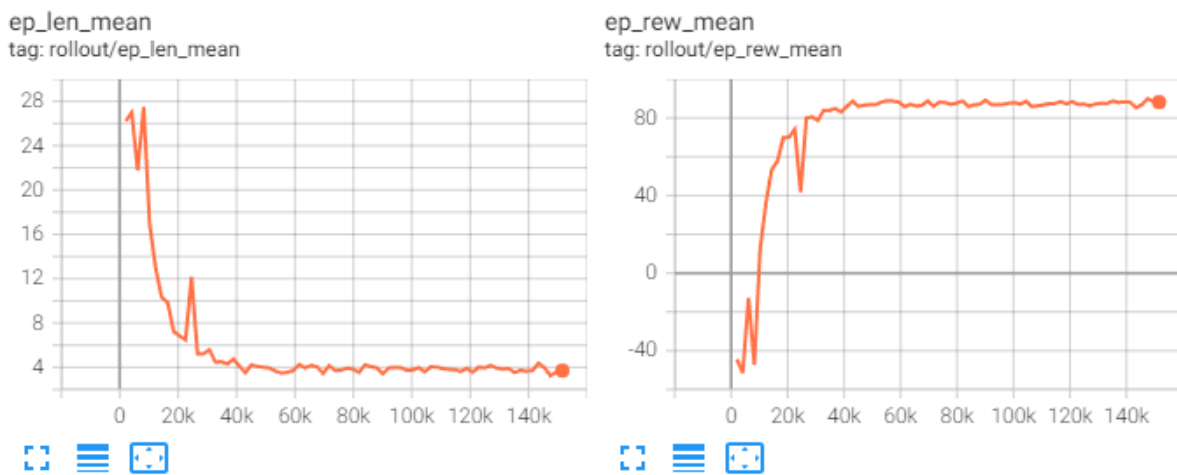


Figure 11: PPO with Gamma 0.99

The high Gamma test of 0.99 plateaued after timestep 26,620 which is impressively fast. Having high Gamma represents a very greedy approach where once information is learned it is very rarely unlearned which explains the rapid climb to the minimum reward range for plateau consideration of 79.98. The maximum for this range was 85.93 for a total range of 5.95.

The convergence rate for Gamma 0.99 was:

$$(79.98 - -44.45) / 26,620 = 0.0046743 \sim 0.00467.$$

Even with the higher starting reward, the rapid increase in reward was so steep that it outpaced Gamma 0.95 and learning rate 0.0001 even with a suboptimal starting point for the convergence rates calculation.

After plateauing, there was a maximum reward of 89.86 at timestep 147,500 and a minimum reward of 85.41 at timestep 143,300 representing an after-plateau range of 4.45, which over the course of around 120,000 timesteps is incredibly low. This shows how much of an effect Gamma has on the variance once an optimal solution has been reached, and the base variance given by the scenario randomly spawning an enemy across from the agent. The variance from this graph shows me that the previous variances were in fact statistically significant as over the course of 2,000 iterations in such a simple environment, I was unsure if the average would be represented by a sufficiently large sample size. 2,000 timesteps seems like an acceptable number of iterations for playing the Basic.py scenario as the variance is so small with high Gamma.

Gamma conclusion

The Gamma tests have shown me the base variance of the Basic.py class once an optimal solution has been reached. For more consistent greedy approaches, a high Gamma is useful. For more complex scenarios, lower Gamma is necessary. Going as low as Gamma 0.9 seems to work well for basic.py, always converging with a learning rate of 0.0001. Adjusting Gamma yielded some of the best convergence rates yet. Perhaps increasing Gamma over time, or for a pre-trained model could be useful. It would cause the agent to become greedier as better paths are discovered early on, but won't learn the oldest paths too well (or will forget them later) so that the agent can learn new, more optimal paths.

I will stick with 0.95 moving forward. Sometimes the agent learns incorrectly and needs the Gamma to be able to climb out of any troughs, but I will use Gamma 0.99 in future for simple, greedy approaches

Gae_Lambda Tests:

Gae_Lambda represents an advantage multiplier. I don't know too much about this variable, so I wanted to experiment with it before moving on to see if it had a substantial impact. I chose to double or half the value and test. I will start with 0.5, then 2.0. I skipped 1.0 because that was the default and already runs great with other settings.

Gae_Lambda 0.5

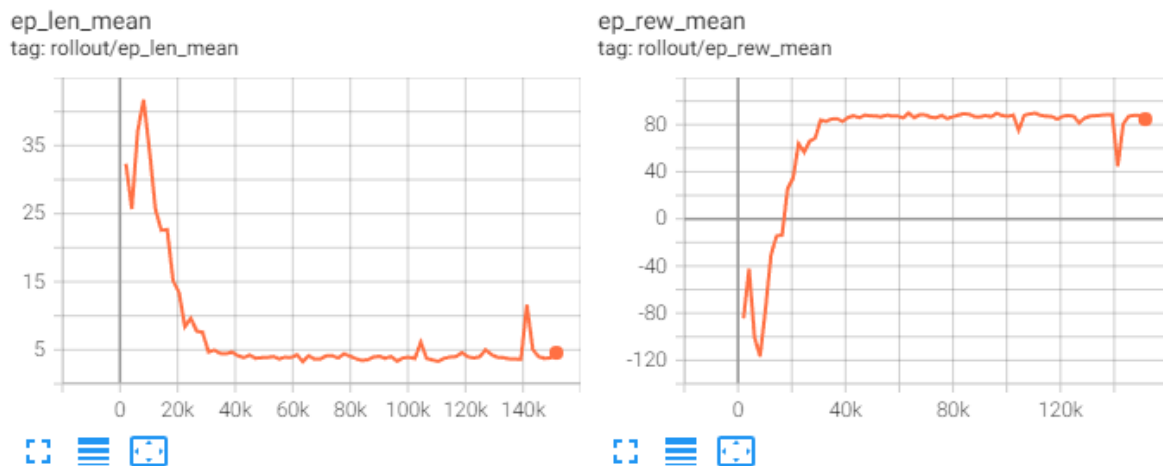


Figure 12: PPO with Gae_Lambda 0.5

For Gae_Lambda 0.5, the graph initially swings wildly around -80 ± 40 for reward until timestep 8,192 where reward increases rapidly towards a convergence point. This graph plateaued with a maximum of 88 to a minimum of 82.4 for a range of 5.6. the plateau was achieved at timestep 30,720 which is very close to where Gamma 0.99 plateaued. At this point, the graph fully converged until a sharp decrease at timestep 141,300 where reward recovered quickly.

The convergence rate of test Gae_Lambda 0.5 from the start was:

$$(83.91 - -84.4) / 30,720 = 0.00580436... \sim 0.00580.$$

The convergence rate of test Gae_Lambda 0.5 from the largest trough was:

$$(83.91 - -116.6) / 22,528 = 0.008900... \sim 0.00890.$$

This convergence rate is 1.5 times the previous best.

The convergence rate of test Gae_Lambda is a new record.

The plateau range for this test had a minimum of 44.77 which was an outlier. Discounting 44.77, the minimum is 75.3. The maximum was 89.57. These values show a plateau range of 14.27 which is notably high. The upper bound was achieved at timestep 96,260. I am not able to tell visually by the graphs what effect Gae_Lambda had on the reward overall or learning dynamic, but it yielded an optimal result when being at 0.5. I need to test if this is an outlier by checking some data points

below and above to ensure this is a sweet spot for the variable, or if there is an even better threshold I can find.

Gae_Lambda 2.0

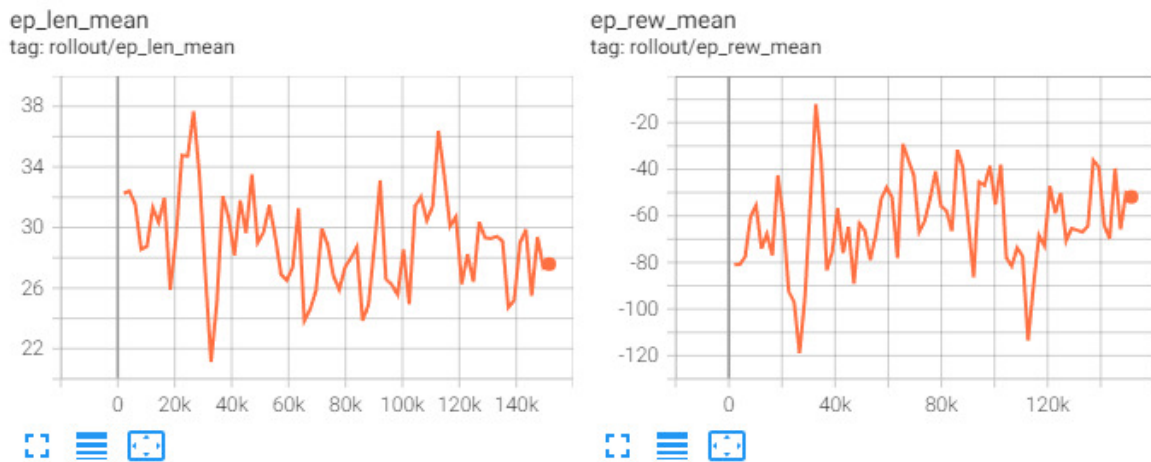


Figure 13: PPO with Gae_Lambda 2.0

For Gae_Lambda 2.0 there was no convergence and reward fluctuated massively between -10 and -120. I am unsure why 0.5 converged and 1 converge but why 2 doesn't, likely a nuance similar to learning rate where being too high causes too much volatility in learning.

Gae_Lambda 0.75

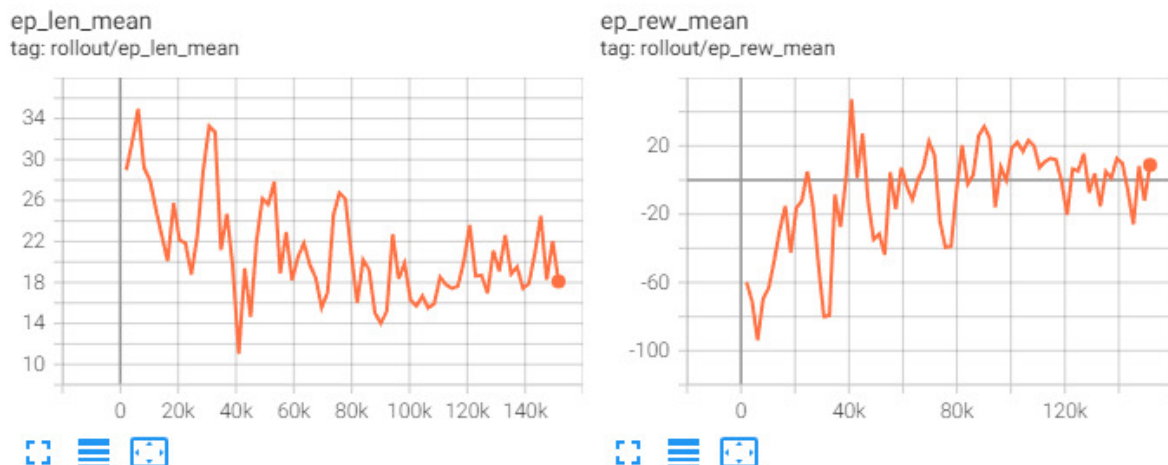


Figure 14: PPO with Gae_Lambda 0.75

Gae_Lambda 0.75 also did not converge which was a surprise. 0.5 and 1 converged. Perhaps given a longer time, it would converge. There is a positive trend seen throughout the reward graph where as more iterations are run, the graph trends upward to around timestep 110,000 where the graph tends back towards reward 0.

Gae_Lambda 0.25

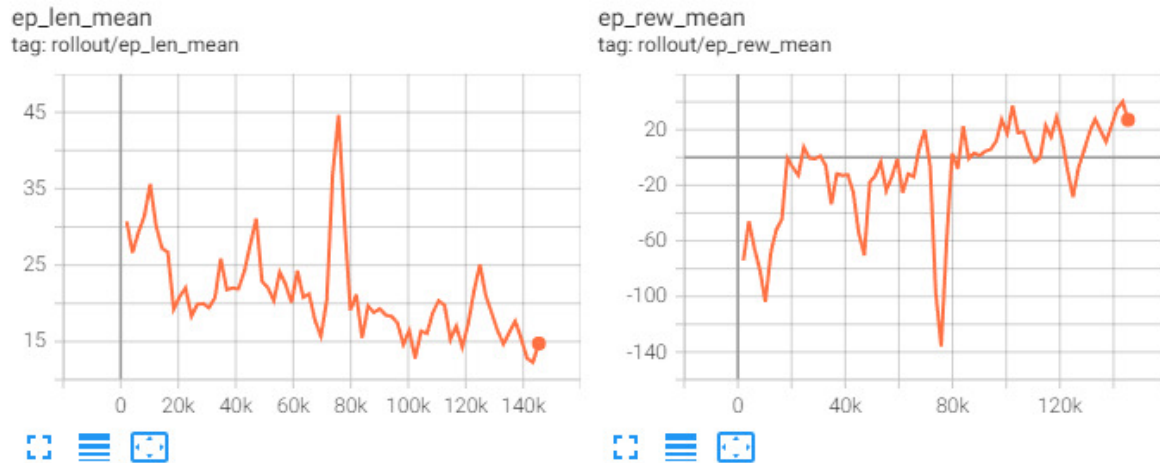


Figure 15: PPO with Gae_Lambda 0.25

Gae_Lambda 0.25 showed an overall positive correlation with a huge sudden spike downwards in reward at timestep 75,780 for unknown reasons. The agent quickly recovered. There didn't appear to be a downward trend present. The graph did not converge making drawing additional results difficult.

Gae_Lambda Conclusion

I think overall lower Gae_Lambda values would be better for the agent in all, though I am unsure where the ideal value lies nor what causes these spikes or trends. Test Gae_Lambda 0.5 had an astonishingly good outcome which makes me consider using it, but further testing shows that this value is likely an outlier due to inconsistency with convergence. Gae_Lambda 1 will consistently converge therefor I will likely continue to use this due to having a time constraint of a few months.

These tests did not indicate to me the relevance of adjusting the advantage multiplier in the Generalised Advantage Estimator.

Basic.py Test Conclusion

The basic.py tests were successful in helping me understand what the more important variables do and have shown me how close to an optimal agent I am for a basic scenario. It has shown me how to use the module correctly, how to adjust config files, and how to consider the complexity of an environment and to adjust the agent accordingly.

I found some good settings such as keeping learning rate minimal at 0.0001 and potentially lower in more complicated scenarios, I might halve this should the deadly_corridor tests struggle to converge. I can see the signs of high variance caused by learning rate being too high as well as gamma being too low, and I can watch the gameplay back to try to find root causes of issues.

Keeping Gamma at 0.95 for general tests will be useful in ensuring the agent converges under general conditions and increasing this up to around 0.99 once optimisation has been completed on curriculum learning and reward shaping to get the environment to meet requirements to converge.

To sum up the tests numerically with successful/positively converging experiments:

Test	Convergence Rate	Plateau range
Learning rate 0.0001	0.00387	14.05
Gamma 0.9	0.00375	15.1
Gamma 0.95	0.00454	16.95
Gamma 0.99	0.00467	4.45
Gae_Lambda 0.5	0.00580	14.27

As seen above, the best test was at Gae_Lambda 0.5 with Learning rate 0.0001 and Gamma 0.95. It didn't have a tight plateau range, but it had a very fast convergence rate. The plateau range was on par with all others except Gamma 0.99 which had an astounding plateau range of 4.45. Gamma is the most adjustable variable to encourage the agent from these 3. This is evident due to 3 successful tests present compared to only 1 from learning rate and 1 from Gae_Lambda. I need to be very careful when adjusting parameters that aren't Gamma. I am surprised that Gamma 0.9 had a plateau range that was lower than Gamma 0.95, since the graph was expected to be so much more volatile.

The settings I will use moving forward will likely be Learning rate 0.0001, Gamma 0.9, and Gae_Lambda 1. I will note if that changes.

Deadly corridor tests:

For deadly corridor I have produced 5 copies of the config file where the difficulty scales from 1 to 5. This represents the difficulty slider in game and adjusts how much damage the enemies do. I will note down the difficulty of the level in the title of the test as "Doom Skill" which is the name the difficulty goes by. The lower the value, the easier the scenario.

Learning rate 0.0001, Gamma 0.95, Gae_Lambda 1 Doom Skill 1:

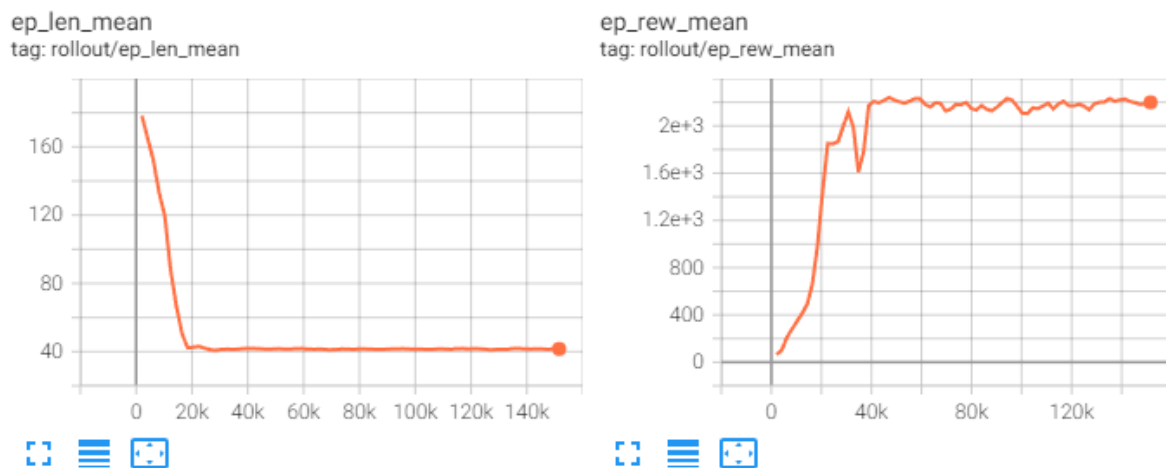


Figure 16: PPO with learning rate 0.0001, Gamma 0.95, gae_Lambda 1 and Doom Skill 1

Surprisingly the agent converged on the first attempt at the easier difficulty and in great time. Preliminary testing of the general deadly_corridor config file showed this result was incredibly difficult to achieve due to the short agent lifetime contrasting with an unchanged amount of timesteps given to the agent to complete it. The agent would need a massive amount of time to beat the scenario at Doom Skill 5. This difficulty however let the agent converge at timestep 38,910 which is great for a more complicated scenario. It shows this agent's model is scalable to higher difficulties.

The convergence rate for this test was:

$$(2172 - -65.81) / 38910 = 0.057512464... \sim 0.05751$$

This is dramatically higher than in the Basic.py tests but it isn't a fair comparison due to the massive difference in total reward. Deadly_Corridor has a much higher possible reward due to the armour giving a huge boost to reward on pickup.

Interestingly, the length almost flatlines at timestep 18,430 but reward is only at 983.8 at this step, with more than half of the optimisation to go. I am curious how the agent manages to optimise the environment that much. I thought there would be a closer correlation but even with a large decrease in reward at timestep 34,820 there is no sign of increase in time taken to beat the scenario. The length graph is also much more stable than the reward once converged.

After convergence there was a maximum value of 2240 at timestep 47,100 and a minimum of 2104 at timestep 102,400. This leaves a plateau range of 136.

Learning rate 0.0001, Gamma 0.95, Gae_Lambda 1 Doom Skill 2:

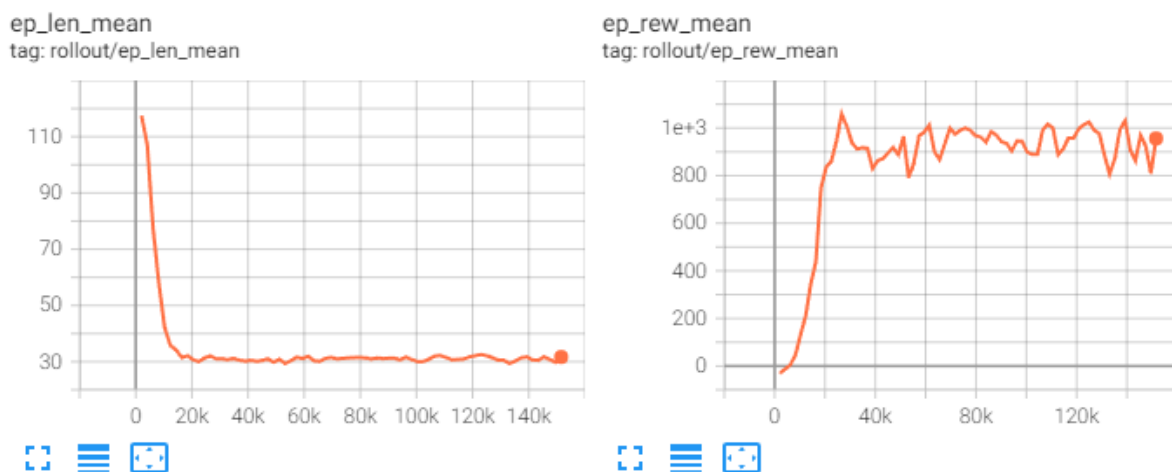


Figure 17: PPO with learning rate 0.0001, Gamma 0.95, gae_Lambda 1 and Doom Skill 2

For this test, the agent somewhat converged but it's hard to say it "plateaued" at all. The volatility in reward is too extreme. This is probably evident of a successful but highly inconsistent strategy. I had to investigate further, so I rewatched the gameplay and found the agent throughout difficulty 1 and 2 running down the middle of the corridor without shooting, hoping to reach the end and acquire the armour pickup.

The convergence rate for the above reward graph was:

$$(937.6 - -31.03) / 30,720 = 0.03151139... \sim 0.03151.$$

This looks much lower than in test 1, but accounting for almost double the reward, I believe this one converged faster. The reason for the reward being doubled in test 1 was due to a bug that I cannot replicate in other tests. I am unsure what causes the issue, but I believe something in the thread at reference [19] could have something to do with it. It claims the living reward was doubled, but in that case, I wonder if other rewards could also be doubled.

Either way, Figure 17 was much more volatile than test 1 even with the double reward bug which is unexpected, and the length graph has noticeable bumps post-convergence. I am going to proceed in future tests checking for this behaviour of the agent learning to run directly to the pickup. The scenario documentation states difficulty 5 is required for this behaviour not to occur.

Learning rate 0.0001, Gamma 0.95, Gae_Lambda 1 Doom Skill 3:

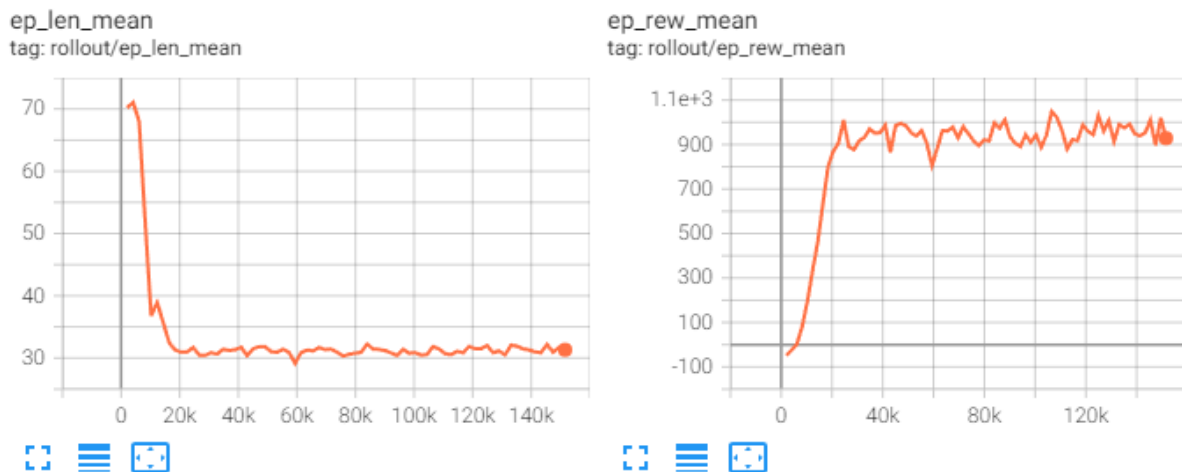


Figure 18: PPO with learning rate 0.0001, Gamma 0.95, gae_Lambda 1 and Doom Skill 3

This agent continued running directly to the armour pickup but due to the difficulty, additional volatility is introduced, and it is becoming more and more apparently. I am not convinced this behaviour will stop persisting, so I will skip difficulty 4 and see how the agent handles difficulty 5.

The convergence rate of this graph was:

$$(1010 - -47.91) / 24580 = 0.043039462... \sim 0.04304$$

While Figure 18 has the fastest convergence rate of any test on deadly corridor so far accounting for the double reward bug, I am unsure how relevant it is due to all the agents doing the same thing. This agent is just faced with a more challenging or rather harmful environment. Visually there seemed to be a convergence overall but there was too much volatility to calculate a reasonable plateau or range.

Curriculum learning and reward shaping

There are some issues when chaining difficulties together, scaling the difficulty, and encouraging constant learning throughout the process.

To encouraging learning, I have implemented new rewards. I have discouraged death, taking damage, and encouraged the agent to hit enemies with their shots. If they miss, they lose reward, and every frame where they idle or don't pick up the armour, the agent loses 1 reward. I believe reward shaping will aid the agent in killing the enemies, which is the implicit goal of this scenario. A substantial problem with default settings on higher difficulties is that the agent will tend to run directly to the armour and ignore the enemies due to no reward, but the agent needs to kill enemies just to reach the armour on Doom skill 5. I am tracking new variables for reward shaping including Health, Damage_Taken, Hitcount, and Selected_Weapon_Ammo. I will use the variables to find

differences in the previous state to the current state, rewarding or penalising the agent for the differences.

Additionally, I will chain the difficulties together, running 100000 episodes per difficulty. 500000 iterations on a longer level should be sufficient for learning. If not, I will load the newest model and continue learning for additional iterations. To chain difficulties together, I will instantiate a new environment and load the previous environment's outputs model data. The agent should be capable of going into difficulty 2 with prior knowledge of difficulty 1 because of the changes I have made. The reward data is tabularised below.

Action	Reward
For moving closer to the armour pickup	+dx
For moving further from the armour pickup	-dx
Picking up armour	+1000
Death	-100
Taking 1 point of damage	-10
Hitting an enemy	30
Shooting	-5
Alive for 1 frame	-1

Curriculum learning/reward shaping Learning rate 0.0001

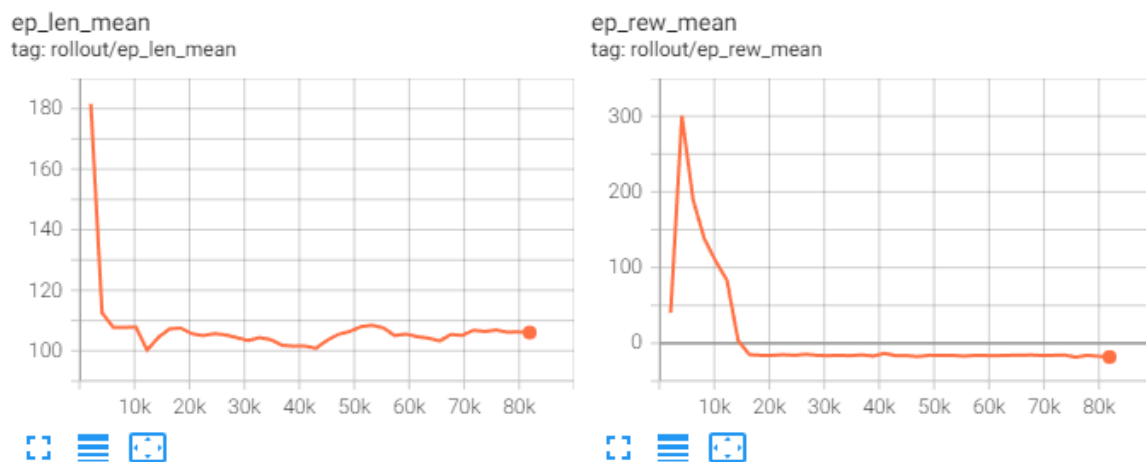


Figure 19: Curriculum learning and reward shaping with PPO, Learning rate 0.0001 Doom Skill 5

Learning rate 0.0001 for the curriculum learning test was not promising as seen in Figure 19. It reminds me of the learning rate 0.0003 in figure 7. Comparing the graphs, there is a large upward spike in reward as the agent applies the understanding of the environment to a higher difficulty when now the enemies can kill the agent in the spawn room. Damage in the Doom Skill 5 scenario is more than the agent is used to, and the stark decrease in reward following implies exploration settings can't be applied because the damage is too punishing. More training is likely required on previous difficulties to ensure a good routine to beating the scenario consistently. I don't know why the lowest reward achieved was -16, when on reviewing the footage, the agent is successfully killing enemies and should be earning a substantial reward. Perhaps the idle punishment is too much, or some penalty is too strong. The length time spent in the environment fluctuates, implying the agent is still operating for a reasonable amount of time given the difficulty. I will conduct further testing on

this methodology to get a better understanding and half the learning rate to encourage lengthier training times in the environments. To compensate for the lower learning rate, I will double the runtime in the environments to 200000 iterations per difficulty.

Curriculum learning/reward shaping Learning rate 0.00005

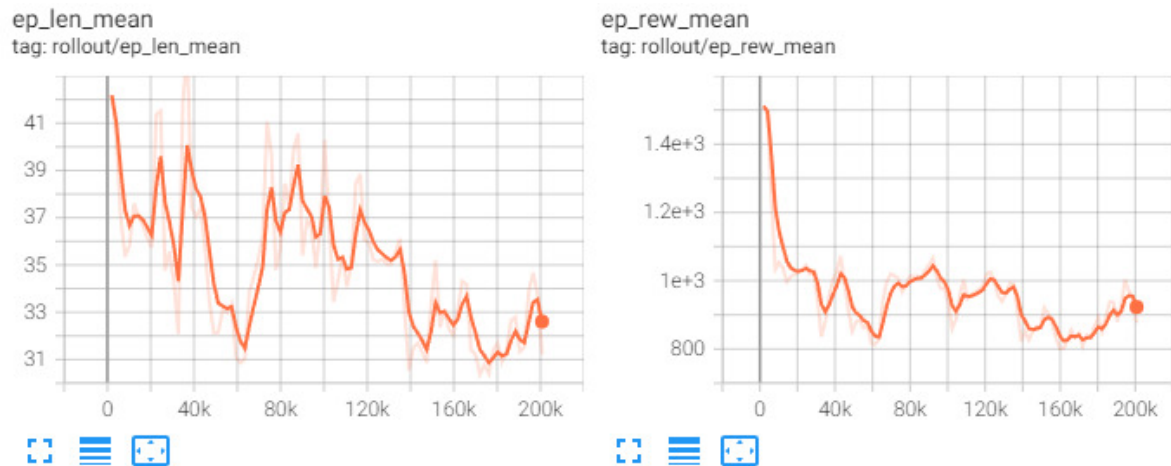


Figure 20: Curriculum learning and reward shaping with PPO, Learning rate 0.00005 and Doom Skill 4

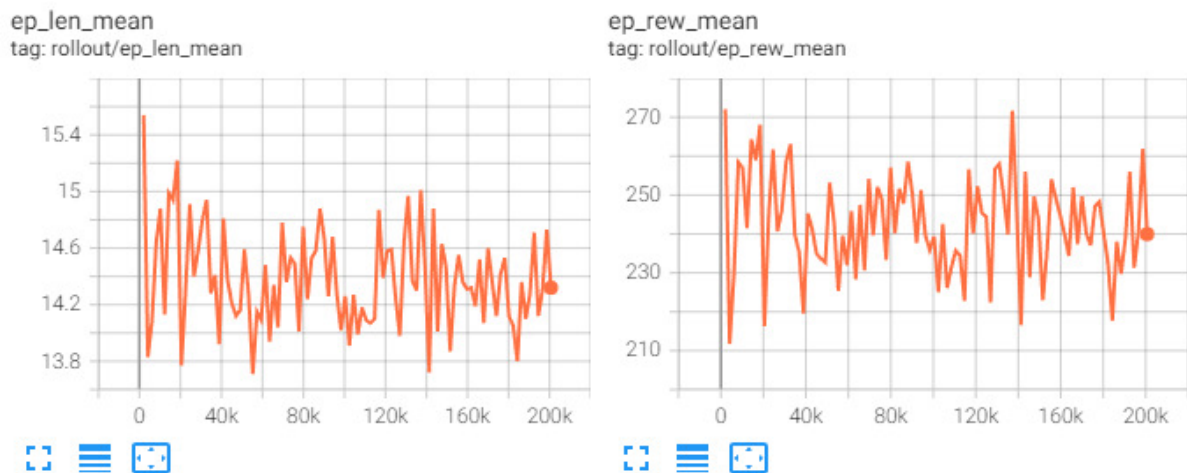


Figure 21: Curriculum learning and reward shaping with PPO, Learning rate 0.00005 and Doom Skill 5

Figure 20 was very promising besides the overall negative reward gradient. It didn't necessarily converge at an optimal solution for the difficulty but became consistent at getting the same mediocre reward over time. The measures to encourage exploration and extend the learning time of the agent were successful given figure 20. Figure 21 shows a somewhat volatile agent having issues handling the higher difficulty, but with more success than the previous agent in figure 19. I believe given more time, an agent could understand the Doom Skill 5 environment enough to beat it and in

fact, the agent did successfully beat it twice in a model test. The inconsistency reveals itself in the low reward comparatively to behind-the-scenes replays. Enemies are killed which was the focus of reward shaping – the fact the agent can kill 2-4 enemies on doom skill 5 difficulty is a testament to the methodology working.

Note the difference of scaling of the y axis in figure 20 to figure 21. The graphs are about as volatile as one another but figure 21 is more defined due to the smaller reward. Over the long term and given the maximum possible reward of more than 2200, the differences here are miniscule compared to the possible reward highs as seen at the beginning of figure 20.

Reward shaping and curriculum learning conclusion

Testing done in this phase showed potential with the current direction, but given context, more code would need to be implemented to give the agent a means to plan, execute the plan, and evaluate why the plan went wrong. I would rather use a new algorithm to do that as with the model-based reinforcement learning algorithms. They handle planning features well.

The agent became able to kill enemies consistently and scored higher reward than previously. The agent had difficulty progressing through Doom Skill 4 and 5 until curriculum learning was implemented, and reward shaping encouraging the agent to attack enemies was vital to the survival on Doom Skill 5, where the agent has stalled for enough time to begin to learn the scenario better, I believe.

There is a bug with the curriculum learning saving system where for some odd reason the save directory is always the lowest difficulty, or the initial saving directory even after a new environment is created with no connection to the old one besides loading model data. I believe somewhere in that model data is a save location which is tripping up specifically the Logs segment which is where the TensorBoard data is saved. This is only a bookkeeping issue and does not affect performance. All data is still saved and is saved in a sequential way, so testing difficulties 1-5 will save to difficulty 1, PPO 1-5 instead of difficulty 1, PPO 1, difficulty 2, PPO 2.

Unfortunately, the bug above has made gathering information on the test from figure 19 quite difficult as it is buried with the pre-curriculum learning tests. I cannot find the correct test to bolster this conclusion with statistics.

Future Work:

Given more time, I would add components such as enemy identification aids (viewing enemy's obvious features, such as yellow eyes, to encourage the ai to look in their direction and shoot them, perhaps in basic.py) which would overall improve the rate of reward gain and improve success rate.

There is potential for some sort of scoreboard to be added such that different algorithms can be more directly compared in some sort of GUI perhaps, with inbuilt graphing using the TensorBoard data. Doing so would avoid the tedious task of opening command prompt at the location, running TensorBoard, and then only being able to view a single test. With this new GUI, I would be able to compare multiple tests overlapping onto the same graph to get a better visual understanding of what changes did what over the long term.

The GUI could also house a leader board for these algorithms or even players, which the algorithm can spectate and learn from initially to improve their own play or go into an environment with prior knowledge.

Spectating is also a function I should have used much more for the complicated scenarios. I would like to implement a feature where I could toggle a spectate mode. Then, I could play DOOM while the algorithm studies what I am doing and why, what reward I obtain for doing various tasks, and go on that basis. I imagine it wouldn't be too useful for simple tasks such as basic.py, but for more complicated ones like Deadly_Corridor, it could be integral to learning rapidly at higher difficulties.

My sample size of algorithms feel's quite small too. Only being able to use A2C and PPO felt like a bottleneck when I began this project, and I didn't want to allocate time to get DQN working. Perhaps it is simpler than I imagined. I think with the short time I have of around 4 months, there was so much uncertainty in the project it was not feasible to work on DQN but following background research, there are plenty of other algorithms I could have tried. Perhaps I should have searched more online for alternative modules to stable baselines 3. I could also try actor critic approaches as they generally have a better record at dealing with visually learning games, according to Langbein.

Planning pairs of hyperparameters in tests could be useful in optimising towards reward, and understanding the additional data provided by TensorBoard could be integral to discovery. Creating more extensive tables, refining the steps between test data, and better analysis on these generated tables could be useful for extending the project into increasingly complicated and difficult environments, even scaling difficulty higher than usually possible in the ViZDOOM environment.

Further pushing reward could be done with a model-based approach or developing some sort of memory system to allow the agent to plan or know where enemies are before the agent gets sight of the targets.

Conclusions:

In this project, I have taken components of Renotte's solution and searched the hyperparameter space for optimal settings for ViZDOOM. I have found settings that have quick learning times, settings that provide stable learning, and tried to find through trial and error a sweet spot in the trade off the two.

Using the algorithms given by Stable Baselines 3 [extra] and Renotte's solution, and with semi-optimised parameters, the agent could learn the more complicated environments that come with ViZDOOM to a lesser extent. Implementing reward shaping and curriculum learning proved to be integral to the learning process.

The approach adopted in this report is a sustainable methodology that could be applied to other reinforcement learning optimisation problems to discover potential hyperparameters, though a less trial and error approach would be more beneficial alongside this one. I believe that tracking how the changes in hyperparameters adjust the rate of convergence and sustainability singularly or in pairs, tabularising the data, and analysing that table could be interesting for parameter optimisation.

Reflection on Learning:

I have learned so much in this project. I have undergone a deep dive into machine learning and challenged what I have learned over the last 10 years of personally looking at, but never getting into, machine learning. I have fixed misconceptions, widened my horizons, learned what algorithms exist and what is and isn't possible, found content that I could potentially study for the next few years if I wanted.

I have been through a full research segment on such a limited time scale that it really pushed me to find relevant information quickly, but also due to the topic being quite high level for me, I needed

that information to be high quality. I needed it explained to me in high level terms such that I can get a vague understanding of the overarching process before going a level deeper. It was challenging but enlightening into the world of machine learning.

When it came to coding, it was overwhelming with how much was required of me. I thought I had so many components I had to produce before I would have an even remotely functioning solution. I found out that not so much was needed, as it was provided by the ViZdoom environment, but this project has given me insight into what the components are for such a project in future. The project has also excited me for doing a similar thing in future for more complicated games. For instance, does this same system work on say Bloons TD5? That could be another interesting experiment as it's a much more open-ended game. Can such an AI develop a strategy that can consistently beat it? How will it handle placement? There are a lot of interesting questions, and perhaps it would force me to develop my own environment. The potential to deepen my skills in this field is huge since ViZDoom was such a generous library handling so many components for me. If the components weren't handled for me, I was very lucky that Renotte produced a video describing the rest. His work and the work of the Marek Wydmuch with the ViZDoom team were integral to this project.

The project taught me how to manage my time around my mood more effectively too. Sometimes I have a really difficult time finding motivation to work and unfortunately it occurred during this project for a variety of reasons. Luckily because of the time plan I had made and was advised to leave additional time for things to go wrong in, I was able to recuperate from the downtime I had and continue when I felt better. That's not a feature you can find anywhere else besides at university. The safety net was nice to have and is a reminder to me that I need to watch how much I work initially on a project to avoid burnout and demotivation, but also how to properly handle my downtime so I can return to work in a reasonable amount of time.

I don't like that I have just said "this is an outlier" for some data on the graphs. Visually, I can verify it, but I know there are tests I can do to check but I didn't have time. In future, I would like to test for significant outliers.

I need to work on the way I structure my reports. I felt like doing writing while running code made my report become more of a work journal than a report. The problem is complicated to deal with since either I can leave the work until the end, which sounds awful for the possible time management issues I could face, or I could restructure the whole document as I write which is too much work.

References:

- [1] The Ultimate DOOM game link https://doom.fandom.com/wiki/The_Ultimate_Doom
- [2] M Wydmuch, M Kempka & W Jaśkowski, ViZDoom Competitions: Playing Doom from Pixels, IEEE Transactions on Games, in print, <https://arxiv.org/abs/1809.03470> *Background: Neural Network Models* [last accessed 4 May 2022]
- [3] ZDOOM and GZDOOM teams, ZDOOM, Available at: <https://zdoom.org/index> and <https://github.com/coelckers/gzdoom> [last accessed 4 May 2022] page 2
- [4] Michael Kempa, ViZDOOM Tutorial, Available at: <http://vizdoom.cs.put.edu.pl/tutorial#short> [last accessed 5 May 2022] page 2
- [5] Josh Achiam and Pieter Abbeel, *Part 2: Kinds of RL Algorithms*, Available at: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html [last accessed 5 May 2022] page 5
- [6] Josh Achiam and Pieter Abbeel, *Part 3: Intro to Policy Optimization*, Available at: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#deriving-the-simplest-policy-gradient [last accessed 5 May 2022] page 5
- [7] Robert Moni, *Reinforcement Learning algorithms — an intuitive overview*, Available at: <https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc> [last accessed 5 May 2022] page 5
- [8] Volodymyr Mnih et al, *Asynchronous Methods for Deep Reinforcement Learning*, Available at: <https://arxiv.org/pdf/1602.01783.pdf> [last accessed 10 May 2022]. Pseudocode for A2C/A3C on page 14. Page 6
- [9] John Schulman et al, *Proximal Policy Optimization Algorithms*, Available at: <https://arxiv.org/pdf/1707.06347.pdf> [last accessed 10 May 2022] Pseudocode for PPO on page 5. Page 6
- [10] Josh Varty, *A step-by-step look at Alpha Zero and Monte Carlo Tree Search*, Available at: <https://joshvarty.github.io/AlphaZero/> [last accessed 5 May 2022]. Page 6
- [11] Josh Achiam and Pieter Abbeel, *Part 1: Key Concepts in RL*, Available at: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#:~:text=Value%20Functions,-It's%20often%20useful&text=By%20value%2C%20we%20mean%20the,in%20almost%20every%20RL%20algorithm. [last accessed 5 May 2022]. Page 6
- [12] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, Available at: <https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf> [last accessed 5 May 2022]. Page 6
- [13] Nicholas Renotte, *Build a Doom AI Model with Python | Gaming Reinforcement Learning Full Course*, <https://www.youtube.com/watch?v=eBCU-tqLGfQ> [last accessed 10 May 2022]. Page 7
- [14] MLK, *Top 20 Reinforcement Learning Libraries You Should Know*, <https://machinelearningknowledge.ai/reinforcement-learning-libraries-you-should-know/> [last accessed 10 May 2022]. Page 7

- [15] Araffin and collaborators, *PPO*, <https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html>, [last accessed 10 May 2022]. Page 8
- [16] Chintan Trivedi, *Proximal Policy Optimization Tutorial (Part 2/2: GAE and PPO loss)*, <https://towardsdatascience.com/proximal-policy-optimization-tutorial-part-2-2-gae-and-ppo-loss-fe1b3c5549e8> [Last accessed 12 May 2022]. Page 9
- [17] Edwin Lisowski, *What is entropy in machine learning?*, https://addepto.com/what-is-entropy-in-machine-learning/?utm_source=rss&utm_medium=rss&utm_campaign=what-is-entropy-in-machine-learning#:~:text=Simply%20put%2C%20entropy%20in%20machine,in%20your%20machine%20learning%20project, [last accessed 10 May 2022] Page 9
- [18] Ahmed Elkarashily, *VLSI Placement using Modified Parallel Simulated Annealing*, https://www.researchgate.net/publication/341902041_VLSI_Placement_using_Modified_Parallel_Simulated_Annealing/figures?lo=1 [last accessed 10 May 2022] Page 11
- [19] mhe500, *Incorrect Results from get_last_reward() when Replaying LMP in Mode.PLAYER #412*, <https://github.com/mwydmuch/ViZDoom/issues/412> [Last accessed 12 May 2022]. Page 26